NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

National University of Ireland, Maynooth

MAYNOOTH, CO. KILDARE, IRELAND.

DEPARTMENT OF COMPUTER SCIENCE,

TECHNICAL REPORT SERIES

Testing Guidelines for Student Projects

Stephen Brown and Rosemary Monahan

NUIM-CS-TR-2005-05

Testing Guidelines for Student Projects

Stephen Brown & Rosemary Monahan
Department of Computer Science, NUI Maynooth
June 2005
Technical Report: NUIM-CS-TR-2005-05

Abstract

The guidelines in this report provide a basis for students to test their software projects. It is not intended to be a complete overview of all testing techniques, but provides a framework to guide students in the verification and validation of their work. The focus is on the testing of executable code; code reviews, requirements validation, and design verification are not addressed in this document, but are as important in the development of high quality software.

1. Introduction

Why do testing? Ideally, a software system developed using formal methods would not need testing: but we are not there yet. And even then you probably need to try your software out in the real world before releasing it.

Ideally software being tested should be of reasonably quality. It is generally accepted that a test-and-fix approach does not result in high quality software, and that 'right-first-time' is a more effective way to produce software. So, the various phases of the development process are used to ensure as high a quality of software as possible prior to testing:

- Requirements analysis should ensure that the user testing goes smoothly.
- System design should ensure that the system testing goes smoothly.
- Detailed design should ensure that the unit testing goes smoothly.

There are two main techniques for generating test cases: **black-box** and **white-box**. In black-box testing, the test cases are derived from the specification: therefore, it is more likely to find faults where the code does *not implement* something in the specification. In white-box testing, the test cases are derived from both the code internals and the specification: therefore it is more likely to find faults where the code implements something *not in* the specification.

What is a 'bug'? A developer's **mistake** leads to a **fault** in the code, and results in an **error** during execution. Errors can be detected by testing; 'debugging' is used to locate and fix faults.

2. The three threes:

1. There are essentially three ways to test software: unit testing, system testing, and user testing. Unit and system testing are verification (verifying that the software meets its specification); user testing is validation (validating the software in real-world use).

2. There are essentially three reasons to test software: to find 'bugs', to certify that it meets a standard (e.g. an internet rfc), or to ensure that it performs in the real world.

3. There are three items needed to test software: a specification of what the software is supposed to do, the software itself, and a test environment (ideally this provides for automated execution of the tests).

2.1 The Three Ways

Unit testing is an exercise whereby a 'unit' of code is tested against its specification. This unit may be a function or procedure or method; it may be an object or module; it or it may be a subsystem. The software to be tested is called with selected inputs, the outputs are collected, and these are compared to the expected outputs. If the comparison succeeds, then the test has passed; otherwise, the test has failed.

System testing is an exercise where an entire system is tested against its specification; it is essentially the same as unit testing, except for one crucial point: the system is tested via its external interface. For user applications this will be a user-interface, probably a graphical user-interface (GUI); for servers this will be a networking protocol interface, such as HTTP; and there are other possible interfaces, for example to computer hardware in the case of an embedded system.

User testing is an exercise where an entire system is tested against the user's expectations – an informal specification! The purpose here is to ensure that the features provided by the software really meet the user's needs. The 'user' may be a person, or it may be one or more other computers (for example, in the case of a network server), or it may be a mixture of both. User testing does not say much directly about the quality of the software, for example a scientific user may be quite happy if the software crashes with invalid inputs. Or, even worse, a user may not notice if incorrect results are produced!

2.2 The Three Reasons

How do you find bugs? You test the software with inputs that have a high probability of not working properly. In **black-box** testing this involves boundary cases, complicated combinations and sequences, and invalid or other 'unexpected' inputs. The initial quality of the software can be ensured using equivalence partitions[1] to provide typical inputs. **White-box** testing is essentially targeted at faults, and simple techniques include statement coverage[1], branch coverage[1], and multiple-condition[1] coverage. If you have used a **formal approach**, you can place assertions[1] in your code to check your pre- and post-conditions at run-time.

How do you certify software? Some standards will include a certification specification: inputs and expected outputs from the system. So these are relatively straightforward to execute. Other standards will not include this, and so test cases must be derived from the specification. You use similar techniques to those shown above: but the focus will be on more complete coverage of correct operation, rather than on exceptions and invalid inputs (though these need to be tested: a good specification will cover all possible inputs). Sometimes, certification is provided through real-world and interoperability testing (validation).

How do you ensure the software works in the real world? Easy – you give it to typical users, or install it in one or more typical environments, and see if the users complain. This activity alone, however, is not an adequate measure of software quality. Typical users only occasionally use more obscure features. Often a software design will be validated using rapid prototyping techniques prior to the development of quality software. The feedback from validation is normally subjective and qualitative; whereas the feedback from unit testing and system testing is normally objective and quantitative.

[1] See the Appendix for details

2.3 The Three Items

What is a specification? It is a clear, unambiguous, and ideally short, definition of what the software is supposed to do. A complete specification will cover all error cases, invalid inputs etc. An incomplete specification will either ignore these, or require some interpretation of what is valid (see the UNIX man pages for examples of this). To make sense, the specification must be produced as a design activity, before the code is written. UML is one 'language' you can use for specifying software operation. The best way of confirming that you have written a good specification, is to see if you can easily derive test cases from it.

Where does the software come from? It is developed from the specifications. Code written with no requirements analysis, and no design specifications, is not likely to be of high quality. In real terms, you take the software specification, then design the code, and then write the code. For small and simple segments of code, this is often compressed into a single activity, with the design held in your mind, and expressed in comments; but not for an entire project…

What is a test environment? Some test automation tools you might use are CppUnit or JUnit for unit testing; and Phantom or Abbot for system (GUI) testing. User testing/validation will usually be manual, though you may write special test tools to automate this: for example, measuring the reliability or accuracy of results produced by a vision system etc. Where do the tests come from? well you develop these from the specifications. In black-box testing, you derive both the inputs and the outputs from the specification: in white-box testing you (usually) derive the inputs from the code, and the outputs from the specification.

3. Testing in the Final Year Project

Here are 10 specific ideas for testing your final year project software:

1. be systematic in your approach to testing – you won't have time to test everything, so make reasoned decisions about what to test, and what level of detail to test it to;
2. develop unit tests for code in parallel with developing the code; mainly use equivalence partitioning, but add stronger tests for important functionality;
3. integrate these unit tests into your build environment: ideally use 'make' to automatically run unit tests at compilation time;
4. use an automated tool to test your system: develop tests from the specifications. If you use 'record/playback' to generate test cases, then verify every recorded output against the specification;
5. mainly use black-box testing – reserve white-box testing for complex/critical code;
6. when you update or change some software in your system, and before you test the new or changed software, you should rerun existing tests to make sure you haven't broken anything - this is called **regression testing.** You don't need to do this immediately after every change: batch them into periodic 'baselevels';
7. use a test coverage tool to measure your code coverage – ideally you should achieve 100% statement coverage and 100% branch coverage. Use tools such as gcov (C/C++) and Emma (Java) to measure this;
8. document your test cases and your test data: one example of how to do this is provided in the E-STD (Tech. Reports NUIM-CS-2002-04 and –05);
9. document your test results: ideally for each build/baselevel of your system;
10. if you are developing a server, or similar software, use performance test tools to verify its operation under load (also referred to as stress testing).

Remember testing fits into any software development approach/process:

- waterfall and incremental development: testing is an integral step – remember to run regression tests to make sure you haven't broken anything when you develop a new version of your software.
- XP: testing is integral and continuous.
- Formal development methods: these provide excellent specifications for testing, and especially with assertions.
- Cleanroom development: testing, and quality measurement (Mean Time to Failure using typical usage profiles) are integral parts of this method.
- Prototyping: remember, that this is generally used to *clarify* requirements, and *not* to produce final code. So, the specification developed by prototyping can be used to develop tests.
- Spiral model (note - this is very difficult to use properly): testing is an inherent part of risk assessment.

See the attached appendix for a very quick description of some testing techniques, some important terminology, and some notes on general testing procedure.

4. Further Information

The best summary of testing methods is in Copeland's book. Another good book to read that covers somewhat more research and principles behind testing is Roper, which provides a good explanation of the basic testing methods discussed here (Roper Ch.3). Note: I recommend using Truth Tables only and avoiding Cause-Effect Graphs and Decision Tables for combinational testing. Another good book, which is somewhat more applied, is Myers – make sure to use the second edition. There are a number of books on OO testing (Marick, Binder, McGregor & Sykes) but they are all rather large. Two useful techniques, however, are combinational testing (Binder Ch.6) and state-machine testing (Binder Ch.7). Also see Testing under Schedule Pressure (Marick Ch.15).

References

- Beck, eXtreme Programming eXplained, Addison Wesley, 2000
- Binder, Testing Object-Oriented Systems, Addison Wesley, 2000
- **Copeland, A Practitioner's Guide to Software Test Design, Artech House, 2004**
- Delaney & Brown, "Document Templates for Student Projects in Software Engineering", NUIM-CS-TR2002-05
- Delaney & Brown, "Guidelines for Documents Produced by Student Projects in Software Engineering", NUIM-CS-TR2002-06
- Myers, The Art of Software Testing, 2nd Edition, Wiley, 2004
- Marick, The Craft of Software Testing, Prentice Hall, 1995
- McGregor & Sykes, A Practical Guide to Testing Object-Oriented Software, 2001
- Roper, Software Testing, McGraw Hill, 1994

APPENDIX – A Very Brief Description of Testing

1. A Very Quick Guide to Black Box Testing (**BBT**):

   Equivalence Partitioning (EP) - for each input and output parameter to a function/method/procedure, identify simple 'partitions' and pick a value in each (not at the boundary). Create the smallest number of test cases such that each parameter has at least one value from each partition. Then test invalid parameter values/partitions one at a time (don't have multiple invalid parameters in the same test).
   Boundary Value Analysis (BVA) - as for EP, but instead of just one typical value in each partition, test for two values: the min & the max. Visualise this as testing all the corners of a square (for 2 parameters), or of a cube (for 3 parameters) etc.
   Combinational Testing/Truth or Decision Tables - draw up a truth-table, and then create a test case for each rule/column. This tests the interaction between parameters.

2. A Very Quick Guide to White Box Testing (**WBT**):

   Statement coverage - generate tests so that every statement is executed.
   Branch Coverage - generate tests so that every branch is executed.
   Multiple-Condition Coverage - generate test data to ensure that every condition in every decision is executed with the values true & false

3. A Very Quick Guide to Assertions:

   Assertions (you can enable/disable these at compile time) – use these to test pre- and post-conditions. Trivial example: at the state of a method which has the precondition "`x>=0`", use "`assert( x>=0 );`" – execution will halt if the condition is not true.

4. A Very Quick Guide to OO Testing:

   Methods – test each method using BBT and/or WBT cases as above
   Objects – use combinational testing between different class methods
   State Machine Testing – if an object has an associated state machine, then test each individual transition: put the software into the required initial state by generating required 'events', generate the test 'event', and check that correct actions have taken place, and that the correct final state is entered. Check implicit & explicit transitions. Note: testing for the final state may require you to add a method to read the state; checking that actions take place is much more difficult!
   UML-diagram based testing: generate test cases/data from the following diagrams: Use Cases, Sequence Diagrams.

5. A Very Quick Guide to GUI Testing:

   Traverse the interface – enter (and exit) every window in the interface.
   Use BBT on each window – use BBT techniques on each input and output component (i.e. parameter) of each window in the interface.
   Task-based Testing – identify the user tasks that the system supports via the interface, and then use BBT techniques on the parameters associated with each task (these will probably involve using many windows). This checks that the application works between windows – combinational testing provides fairly thorough testing.

6. Testing terminology:

    a. **Bug**: an *error* in the programmers mind, causes a *fault* in the code, which results in an *failure* during execution
    b. **Partition**: if different values of a parameter cause different processing to occur, then the parameter has multiple value 'partitions'. For example, if the specification states that different processing occurs for positive and negative integers, then you have two partitions. If it states a parameter must be positive, but the type is "`int`", then you still have two partitions: positive (valid), and negative (invalid).
    c. **Boundary value**: each partition has two boundary values: one at the top (the maximum), and one at the bottom (the minimum).
    d. **Truth-table**: a boolean table that relates the outputs to the inputs. Here is a trivial example: note that n input conditions *can* give a max of to 2^n rules.

	Rules		
INPUT CONDITIONS	1	2	3
x>0	T	F	F
x<0	F	T	F
OUTPUT CONDITIONS			
return value=x	T	F	T
return value=-x	F	T	F

    Note that we do not need to explicitly list the condition x==0.
    You can use * for "don't care" input conditions – but never for outputs.
    (The decision table and cause-effect graph are other forms of the truth table).
    e. **Condition**: in the code "`if (x==10) y = 14;`" the decision has one condition: "`x==10`".
    f. **Multiple conditions**: in "`if ((x==10) && (y<0)) y = 14;`" the decision has multiple conditions: "`x==10`" and "`y<0`".

7. General Procedure for Testing Software:

    a. generate test cases (using one of the techniques shown here)
    b. generate test data: identify input values, and expected output values – sometimes you can exercise several test cases with one set of data
    c. generate tests: use unique id's (sequential numbers, or use a numbering hierarchy) and document which test case(s) each test covers
    d. for automated tests, implement the tests: code up the tests (using suitable naming and comments to identify the test id's and cases covered), or put the test data into a file and use generic test code
    e. execute the tests: provide the input values for each test, and compare the actual output with the expected output
    f. measure coverage: use a tool to measure the total coverage of all the tests on a 'unit' of software
    g. collect & record the results: for example, the date, the software under test (program/object/method name and version number or baselevel), the test pass rate, statement coverage, branch coverage, …
    h. fix faults & rerun the tests